Image Processing Acceleration Techniques using Intel® Streaming SIMD Extensions and Intel® Advanced Vector Extensions

September 4, 2009

Authors: Petter Larsson & Eric Palmer

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY RELATING TO SALE AND/OR USE OF INTEL PRODUCTS, INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, life sustaining applications.

This document, as well as the software described in it, is furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document.

Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

Intel may make changes to specifications and product descriptions at any time, without notice.

The Intel® Viiv™ technology may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents, which have an order number and are referenced in this document, or other Intel literature, may be obtained by calling 1-800-548-4725, or by visiting Intel's Web Site.

Contents

Figures

Tables

Introduction

Modern Intel processors features acceleration through the use of SIMD (Single Instruction Multiple Data) instructions that include a wide range of available Intel® Streaming SIMD Extensions (Intel® SSE) instructions and the new Intel® Advanced Vector Extensions (Intel® AVX) instructions. Image processing data structures and algorithms are often suitable candidates for optimizations using these instruction sets. Combined with the Intel® C++ Compiler's ability to autovectorize loops it provides for an efficient method of achieving improved performance in applications dealing with image processing.

In this paper we will detail some well known transformation techiniques with code examples illustrating how to take advantage of Intel® SSE and Intel® AVX to transform image data, together with compiler autovectorization of image processing algorithms. The paper details optimized implementations (using varying data types and sizes) of data transformations and algorithms together with analysis comparing performance and providing speedup measurements for Intel® SSE optimized code and estimates for Intel® AVX optimized code.

Intel® AVX is a 256 bit instruction set extension to Intel® SSE and is designed for applications that are floating point intensive. Intel® AVX extends all the 16 XMM registers to 256 bits YMM registers, doubling the register width leading to improved performance and power efficiency over the 128 bit SIMD instructions. Use of Intel® AVX also results in fewer register copies, more efficient register use and smaller code size.

Using the proposed techniques we can achieve good performance speedup as can be seen in the performance speedup summary below.

Filter	**Intel® SSE Speedup**	**Intel® AVX Speedup**
Sepia (int base)	2.6x	3.1x
Sepia (float base)	1.9x	2.2x
Crossfade (int base)	2.7x	3.6x
Crossfade (float base)	1.9x	2.4x

Measured for Intel® Core™ i7 processor with recommended chunk size of ~50000 pixels. Note that Intel® AVX performance was estimated using a simulator, and that it does not take into account future architecture improvements.

Overview

The code examples provided in this article assumes use of Intel® C++ Compiler and requires basic knowledge of SIMD, Intel® SSE instruction intrinsics and how to perform auto vectorization. Compiler features, options and pragmas apply to the use of Intel® C++ Compiler 11.1.35 or later which supports new instructions sets such as Intel® AVX.

The code examples are in C++ and were built and analyzed on Microsoft Windows* (Vista and XP).

Scope and assumptions:

- Images are represented by uncompressed RGBA pixel values where each color channel is represented by either an integer (8 bit) or a float (32 bit)
- To simplify conversions, color values are represented by a number from 0 to 255, stored in either 8 bit integer or 32 bit float
- Data is aligned to 16 bytes except for the processing involving use of Intel® AVX where 32 byte alignment is used

Notes on performance/speedup:

- Performance of functions using Intel® AVX are estimates since Intel processors featuring Intel® AVX are not yet available. Architecture emulator (Intel® Software Development Emulator) and simulator (Intel® Architecture Code Analyzer) were used to verify behavior and estimate Intel® AVX performance
- The actual performance depends on processor architecture, cache configuration and size, frequency etc.
- Only a few image processing filters (algorithms) are presented in the paper. Performance speedup applicability to other filters depends on the filter complexity and inter pixel dependencies. There are no guarantees of improved performance utilizing the discussed techniques on other filters

Image Processing Acceleration

This article will focus on two central techniques to improve image processing performance

1. Efficient expansion, compression and transformations (transposing) of image pixel data in the form of Array of Structures (AoS) into data in the form of Structure of Arrays (SoA). Transform to SoA enables the use of SIMD instructions in image processing algorithms for improved performance. The transformation computation overhead will in most cases be acceptable due to the gain in algorithm performance.

2. How to develop image processing algorithms allowing efficient autovectorization using Intel® Compiler for optimized performance on Intel® SSE and Intel® AVX architectures. The compiler autovectorization feature detects code that can be optimized (parallell data operations) by utilizing the range of available SIMD instructions in Intel® SSE and Intel® AVX.

Our goals are to provide a number of optimized utility functions to perform expansion, compression and transformations as described above, and also to allow the developer of the image processing filter (algorithm) to easily take advantage of compiler auto vectorization (into Intel® SSE or Intel® AVX instructions) without having to rely on use of SIMD instrinsics or hand coding SIMD assembler code.

Similar transformations are also useful in the realm of some 3D graphics algorithms where coordinates are structured in sets of X, Y, Z vectors.

The following filters will be analyzed due to simplicity and autovectorization suitability. Both of the filters opererate on the image in a pixelwise manner (surrounding pixels are not sampled)

- Sepia: Applies color toning making the image look "old"

$$\begin{aligned} R_s &= 0.393 * R + 0.769 * G + 0.189 * B \\ G_s &= 0.349 * R + 0.686 * G + 0.168 * B \\ B_s &= 0.272 * R + 0.534 * G + 0.131 * B \end{aligned}$$

- Crossfade: Merges two images into one with selected intensity

$$C_c = (1 - I) * C_1 + I * C_2$$

(where C_x = color plane of pixel in image x, I = crossfade intensity)

We will analyze how to achieve image processing acceleration using uncompressed (raw RGBA) image data where the pixel color planes are represented by either 8 bit integer or 32 bit float data types.

Data structure transformations

Transpose

A common way of representing the pixels of an image is through the RGBA color channels, where RGBA translates into the Red (R), Green (G), Blue (B) and Alpha (A) channel.

R	G	B	A

Pixels structured in this form are not very suitable for auto vectorization. This is illustrated by grouping four pixels together into an Array of Structures (AoS). For instance, a 128 bit wide SIMD instruction such as MULPS (_mm_mul_ps), operates 4 single precision consecutive float values. A multiply operation on a color channel of a pixel can not take advantage of MULPS since the pixel values for each color are not consecutive in memory.

R_0	G_0	B_0	A_0
R_1	G_1	B_1	A_1
R_2	G_2	B_2	A_2
R_3	G_3	B_3	A_3

By transposing the above AoS group of pixels into pixel data ordered by color channel, represented by the Structure of Arrays (SoA) structure below, we open up the ability for improved algorithm performance via use of SIMD (through autovectorization, use of intrinsics or hand coding). For instance, the MULPS instruction can now operate on 4 pixels at a time, when multiplying pixel color channels.

R_0	R_1	R_2	R_3
G_0	G_1	G_2	G_3
B_0	B_1	B_2	B_3
A_0	A_1	A_2	A_3

Note that by nature the transpose operation is self inverse, so when applied again on the SoA above it will transpose the matrix back to the original form, an AoS structure.

By using the new Intel® AVX instructions we are able to extend the transposition to 8 consecutive color plane values to allow for efficient use of the new Intel® AVX SIMD instructions operating on 256 bit wide registers (YMM and XMM), compared to Intel® SSE SIMD that operates on 128 bit registers (XMM).

R_0	G_0	B_0	A_0
R_1	G_1	B_1	A_1
R_2	G_2	B_2	A_2
R_3	G_3	B_3	A_3
R_4	G_4	B_4	A_4
R_5	G_5	B_5	A_5
R_6	G_6	B_6	A_6
R_7	G_7	B_7	A_7

⇨

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7
G_0	G_1	G_2	G_3	G_4	G_5	G_6	G_7
B_0	B_1	B_2	B_3	B_4	B_5	B_6	B_7
A_0	A_1	A_2	A_3	A_4	A_5	A_6	A_7

In this paper we are providing optimized techniques for efficient transformations from AoS to SoA pixel data and the reverse to convert back to format suitable for rendering. Refer to Appendix A for details.

Expansion and Compression

In this paper we are also providing optimized techniques for expanding a data structure represented by 8 bit integer to 32 bit float, and the reverse operation of compressing 32 bit float to 8 bit integer. Refer to Appendix A for details.

Data structure transformation implementation

The following table lists all the transformations functions used for performance analysis comparisons. The functions are designed to be very efficient to minimize the overhead of converting from one structure to another. There is a significant performance speedup compared to scalar implementations of these transformation functions.

The efficiency, captured using Intel® Architecture Code Analyzer tool, is indicated in the table as instruction throughput and data dependency latency (based on Intel® Core™ i7 processor architecture). The data refers to the function loop core operating on 4 or 8 pixels at the time. Note that the Intel® AVX instruction efficiency is an estimate.

Function name	Instruction throughput	Data dependency latency	Instructions/ pixel
Intel® SSE optimized			
`transposeFloat`	12	10	3
`expandIntToFloat`	5	12	1.25
`compressFloatToInt`	5	13	1.25
`transposeAndExpandIntToFloat`	5	12	1.25
`transposeAndCompressToInt`	5	13	1.25
Intel® AVX optimized			
`transposeFloatAVX (&reverse)`	12	12	1.5
`expandIntToFloatAVX`	9	11	1.125
`compressFloatToIntAVX`	9	13	1.125
`transposeAndExpandIntToFloatAVX`	9	11	1.125
`transposeAndCompressToIntAVX`	9	13	1.125

Table 1: Data structure transformation throughput

As can be seen from the table the Intel® AVX versions of the functions are very efficient compared to Intel® SSE versions due to the use of the new wider instruction set. This is especially visible from throughput of "transposeFloatAVX" which processes a pixel using half the number of instructions.

Note that Instruction throughput above is the optimal case and assumes that data is cache resident and no other conflicts occur such as cache misses or bank conflicts etc.

Please refer to Appendix A for implementation details for each function listed above.

Image processing filters

In this paper we focus on implementing the image processing algorithms (filters) as a kernel, which operates on a pixel by pixel basis. The complete framework detailing how to process an image through a kernel is out of scope for this paper.

The filters are designed to operate on float data for best precision and performance. Three different algorithm types are compared, for which the pixel data is represented by the following data structures: (refer to Transpose chapter for a graphical illustration)

Pixel **(ordered by AoS)**	**SoA group of 4 pixels** **(used by Intel® SSE algorithms)**	**SoA group of 8 pixels** **(used by Intel® AVX algorithms)**

`typedef struct {`
`        float red;`
`        float green;`
`        float blue;`
`        float alpha;`
`} CData32;`
	`typedef struct {`
	`        float red[4];`
	`        float green[4];`
	`        float blue[4];`
	`        float alpha[4];`
	`} CData32T4;`
		`typedef struct {`
		`        float red[8];`
		`        float green[8];`
		`        float blue[8];`
		`        float alpha[8];`
		`} CData32T8;`

Table 2: Data structures

The compiler pragma "ivdep" is used to direct the compiler to autovectorize the loop (to ensure vectorization even if flow dependence confuses the compiler).

The filter below uses two utility functions, "GetCurrentSrcPointer<Type>" and "GetCurrentOutputPointer<Type>", to fetch the current pixel offset from the framework driving the kernel filter processing. This implementation assumes a filter that operates on individual pixels, while a more advanced filter might require sampling of pixels surrounding the current processed pixel (out of scope for this paper).

The "__assume_aligned" intrinsic is used throughout the filter implementations to convey to the compiler that proper memory alignment is used allowing the compiler to use optimal instructions for moves etc.

Sepia filter

The Sepia filter, as described in the start of this chapter, applies Sepia toning to an image. Since the valid values are in the range of 0-255 represented by float, and due to the nature of Sepia algorithm used it may produce value exceeding 255. Therefore each pixel color value has to be truncated if outside the specified range.

Sepia filter operating on image represented by AoS data

```
void SepiaFloatKernel()
{
        CData32 *pSrc = m_pEngine->GetCurrentSrcPointer32();
        CData32 *pDst = m_pEngine->GetCurrentOutputPointer32();
#ifdef __INTEL_COMPILER
```

```
    __assume_aligned(pSrc, 16);
    __assume_aligned(pDst, 16);
#endif
        pDst->red   = 0.393f * pSrc->red + 0.769f * pSrc->green + 0.189f * pSrc->blue;
        pDst->green = 0.349f * pSrc->red + 0.686f * pSrc->green + 0.168f * pSrc->blue;
        pDst->blue  = 0.272f * pSrc->red + 0.534f * pSrc->green + 0.131f * pSrc->blue;
        pDst->alpha = pSrc->alpha;
        if(pDst->red > 255)   pDst->red = 255;
        if(pDst->green > 255) pDst->green = 255;
        if(pDst->blue > 255)  pDst->blue = 255;
}
```

Alternatively the Sepia filter can be implemented as a selfcontained function as below. Note that the performance of the selfcontained filter vs. the kernel filter, with framework, was verified to be equal.

```
void SepiaFloat(float *pImage, float *pResult, unsigned int imageSize)
{
        CData32 *pSrc = (CData32 *)pImage;
        CData32 *pDst = (CData32 *)pResult;
#ifdef __INTEL_COMPILER
    __assume_aligned(pSrc, 16);
    __assume_aligned(pDst, 16);
#endif
        for(int i=0;i<imageSize;i++)
        {
                pDst[i].red   = 0.393f*pSrc[i].red + 0.769f*pSrc[i].green + 0.189f*pSrc[i].blue;
                pDst[i].green = 0.349f*pSrc[i].red + 0.686f*pSrc[i].green + 0.168f*pSrc[i].blue;
                pDst[i].blue  = 0.272f*pSrc[i].red + 0.534f*pSrc[i].green + 0.131f*pSrc[i].blue;
                pDst[i].alpha = pSrc[i].alpha;
                if(pDst[i].red > 255)   pDst[i].red = 255;
                if(pDst[i].green > 255) pDst[i].green = 255;
                if(pDst[i].blue > 255)  pDst[i].blue = 255;
        }
}
```

Regardless of using compiler pragma or use of internal pixel loop, none of the above filters, operating on image represented by AoS data, gets vectorized by the compiler into SIMD instructions due to non consecutive color plane float values.

Sepia filter operating on image represented by SoA data

The following kernel is constructed to work on 4 pixel at a time to facilitate Intel® SSE auto vectorization.

```
void SepiaTransposedFloatKernel()
{
        CData32T4 *pSrc = m_pEngine->GetCurrentSrcPointer32T4();
        CData32T4 *pDst = m_pEngine->GetCurrentOutputPointer32T4();
#ifdef __INTEL_COMPILER
    __assume_aligned(pSrc, 16);
    __assume_aligned(pDst, 16);
#endif
#pragma ivdep
        for(int j=0;j<4;j++)
        {
                pDst->red[j]   = 0.393f*pSrc->red[j]+0.769f*pSrc->green[j]+0.189f*pSrc->blue[j];
                pDst->green[j] = 0.349f*pSrc->red[j]+0.686f*pSrc->green[j]+0.168f*pSrc->blue[j];
                pDst->blue[j]  = 0.272f*pSrc->red[j]+0.534f*pSrc->green[j]+0.131f*pSrc->blue[j];
                pDst->alpha[j] = pSrc->alpha[j];
                if(pDst->red[j] > 255)   pDst->red[j] = 255;
                if(pDst->green[j] > 255) pDst->green[j] = 255;
                if(pDst->blue[j] > 255)  pDst->blue[j] = 255;
        }
}
```

The above filter, operating on image represented by SoA (transposed) image data, gets autovectorized by the compiler into Intel® SSE instructions MULPS and ADDPS.

The following kernel is constructed to work on 8 pixel at a time to facilitate Intel® AVX auto vectorization.

```
void SepiaTransposedFloatAVXKernel()
{
        CData32T8 *pSrc = m_pEngine->GetCurrentSrcPointer32T8();
        CData32T8 *pDst = m_pEngine->GetCurrentOutputPointer32T8();
#ifdef __INTEL_COMPILER
    __assume_aligned(pSrc, 32);
    __assume_aligned(pDst, 32);
#endif
#pragma ivdep
        for(int j=0;j<8;j++)
        {
                pDst->red[j]   = 0.393f*pSrc->red[j]+0.769f*pSrc->green[j]+0.189f*pSrc->blue[j];
                pDst->green[j] = 0.349f*pSrc->red[j]+0.686f*pSrc->green[j]+0.168f*pSrc->blue[j];
                pDst->blue[j]  = 0.272f*pSrc->red[j]+0.534f*pSrc->green[j]+0.131f*pSrc->blue[j];
                pDst->alpha[j] = pSrc->alpha[j];
                if(pDst->red[j] > 255)   pDst->red[j] = 255;
                if(pDst->green[j] > 255) pDst->green[j] = 255;
                if(pDst->blue[j] > 255)  pDst->blue[j] = 255;
        }
}
```

The above filter, operating on image represented by SoA (transposed) image data, gets autovectorized by the compiler into Intel® AVX instructions VMULPS and VADDPS.

Crossfade filter

The Crossfade filter is a pretty straightforward implementation which blends two images into one resulting image using an intensity factor deciding the strength of each image.

Crossfade filter operating on image represented by AoS data

```
void CrossfadeFloatKernel()
{
        CData32 *pSrc1 = m_pEngine->GetCurrentSrcPointer32();
        CData32 *pSrc2 = m_pEngine->GetCurrentSrcPointer32_2();
        CData32 *pDst =  m_pEngine->GetCurrentOutputPointer32();
#ifdef __INTEL_COMPILER
        __assume_aligned(pSrc1, 16);
        __assume_aligned(pSrc2, 16);
        __assume_aligned(pDst,  16);
#endif
        pDst->red   = (1.0f - m_Intensity)*pSrc1->red   + m_Intensity*pSrc2->red;
        pDst->green = (1.0f - m_Intensity)*pSrc1->green + m_Intensity*pSrc2->green;
        pDst->blue  = (1.0f - m_Intensity)*pSrc1->blue  + m_Intensity*pSrc2->blue;
        pDst->alpha = pSrc1->alpha;
}
```

The above filter, operating on image represented by AoS data, does not get vectorized into SIMD instructions due to non consecutive float values.

Crossfade filter operating on image represented by SoA data

The following kernel is constructed to work on 4 pixel at a time to facilitate Intel® SSE auto vectorization.

```
void CrossfadeTransposedFloatKernel()
{
        CData32T4 *pSrc1 = m_pEngine->GetCurrentSrcPointer32T4();
        CData32T4 *pSrc2 = m_pEngine->GetCurrentSrcPointer32T4_2();
        CData32T4 *pDst =  m_pEngine->GetCurrentOutputPointer32T4();
#ifdef __INTEL_COMPILER
        __assume_aligned(pSrc1, 16);
        __assume_aligned(pSrc2, 16);
        __assume_aligned(pDst,  16);
#endif
#pragma ivdep
        for(int j=0;j<4;j++)
        {
                pDst->red[j]   = (1.0f-m_Intensity)*pSrc1->red[j]   + m_Intensity*pSrc2->red[j];
                pDst->green[j] = (1.0f-m_Intensity)*pSrc1->green[j] + m_Intensity*pSrc2->green[j];
                pDst->blue[j]  = (1.0f-m_Intensity)*pSrc1->blue[j]  + m_Intensity*pSrc2->blue[j];
                pDst->alpha[j] = pSrc1->alpha[j];
        }
}
```

The above filter, operating on image represented by SoA (transposed) image data, gets autovectorized by the compiler into Intel® SSE instructions MULPS, ADDPS and SUBPS.

The following kernel is constructed to work on 8 pixel at a time to facilitate Intel® AVX auto vectorization.

```
void CrossfadeTransposedFloatAVXKernel()
{
        CData32T8 *pSrc1 = m_pEngine->GetCurrentSrcPointer32T8();
        CData32T8 *pSrc2 = m_pEngine->GetCurrentSrcPointer32T8_2();
        CData32T8 *pDst =  m_pEngine->GetCurrentOutputPointer32T8();
#ifdef __INTEL_COMPILER
        __assume_aligned(pSrc1, 32);
        __assume_aligned(pSrc2, 32);
        __assume_aligned(pDst,  32);
#endif
#pragma ivdep
        for(int j=0;j<8;j++)
        {
                pDst->red[j]   = (1.0f-m_Intensity)*pSrc1->red[j]   + m_Intensity*pSrc2->red[j];
                pDst->green[j] = (1.0f-m_Intensity)*pSrc1->green[j] + m_Intensity*pSrc2->green[j];
                pDst->blue[j]  = (1.0f-m_Intensity)*pSrc1->blue[j]  + m_Intensity*pSrc2->blue[j];
                pDst->alpha[j] = pSrc1->alpha[j];
        }
}
```

The above filter, operating on image represented by SoA (transposed) image data, gets autovectorized by the compiler into Intel® AVX instructions VMULPS, VADDPS and VSUBPS.

Performance analysis

By utilizing data structure transformations together with the auto vectorizable filters described in the previous chapter we observe significant speedup compared to the performance of non optimized implementation.

Platform configurations and setup

To identify processor architecture and cache size impact on performance, two system configurations were used.

Intel® Core™ 2 Duo Processor Platform	Intel® Core™ i7 Processor Platform
T7300, 2.0 GHz, 2GB RAM (Dual core, 4 MB L2 cache)	2.66 GHz, 2GB RAM, (Quad core, 4x256MB L2 cache, 8MB L3 cache)

The following table illustrates the compared methods used for algorithm analysis:

Base image element data type	Baseline process	Optimized process Intel® SSE	Optimized process Intel® AVX

8 bit integer	`expandIntToFloat()`
	`<filterAlgorithm>`
	`compressFloatToInt()`
		`transposeAndExpandToFloat()`
		`<transposedFilterAlgorithm>`
		`transposeAndCompressToInt()`
			`transposeAndExpandToFloatAVX()`
			`<transposedFilterAVXAlgorithm>`
			`transposeAndCompressToIntAVX()`

32 bit float	`<filterAlgorithm>`	`transposeFloat()`
		`<transposedFilterAlgorithm>`
		`transposeFloat()`
			`transposeFloatAVX()`
			`<transposedFilterAVXAlgorithm>`
			`transposeFloatAVX_reverse()`

Table 3: Analysis process matrix

Where <filterAlgorithm> is either SepiaKernelFloatKernel() or CrossfadeKernelFloatKernel() and <transposedFilterAlgorithm> is either SepiaKernelTransposedFloatKernel() or CrossfadeKernelTransposedFloatKernel() and <transposedFilterAVXAlgorithm> is either SepiaKernelTransposedFloatAVXKernel() or CrossfadeKernelTransposedFloatAVXKernel(), as described in previous chapter.

Intel® SSE Optimization Results

The following graphs show the relative performance speedup, for the Sepia and Crossfade filter, using optimized Intel® SSE processes versus the un-optimized process for both integer and float base. It is clear that the optimization techniques detailed in this paper provides a good speedup for both of the algorithms studied. The Intel® Core™ i7 processor platform provides, aside from better overall performance, a greater optimal chunk size range compared to the Intel® Core™ 2 Duo processor platform. Also, as can be seen from the graph, the average speedup is greater on an Intel® Core™ 2 Duo processor-based system, the reason for this is due to architectural improvements in Intel® Core™ i7 processors improving non optimized code performance.

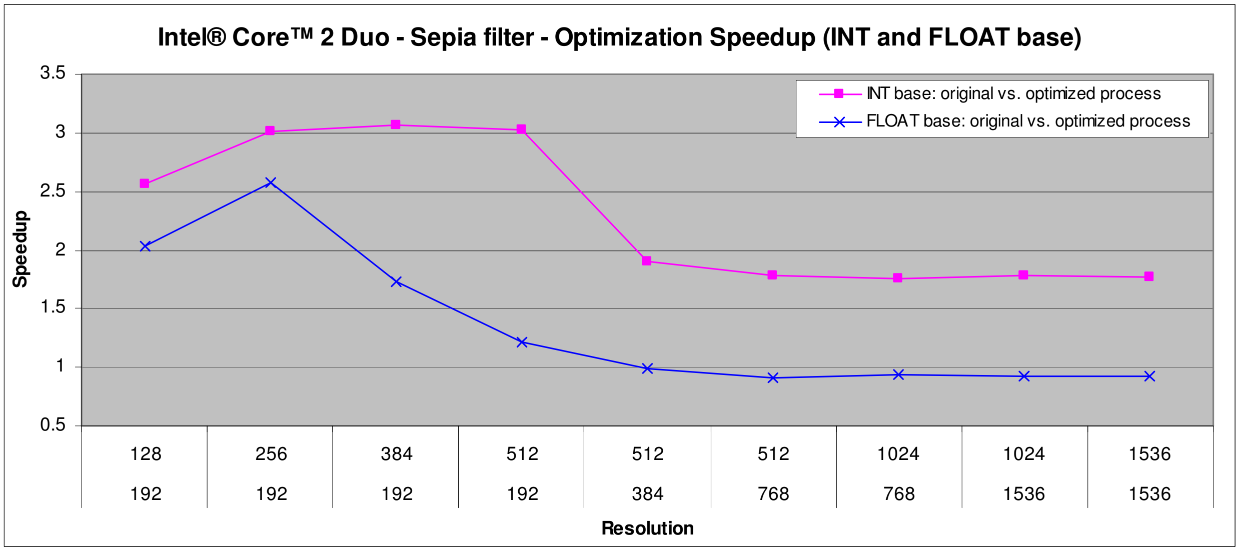


Figure 1: Sepia filter performance comparison. Intel® Core™ Duo processor

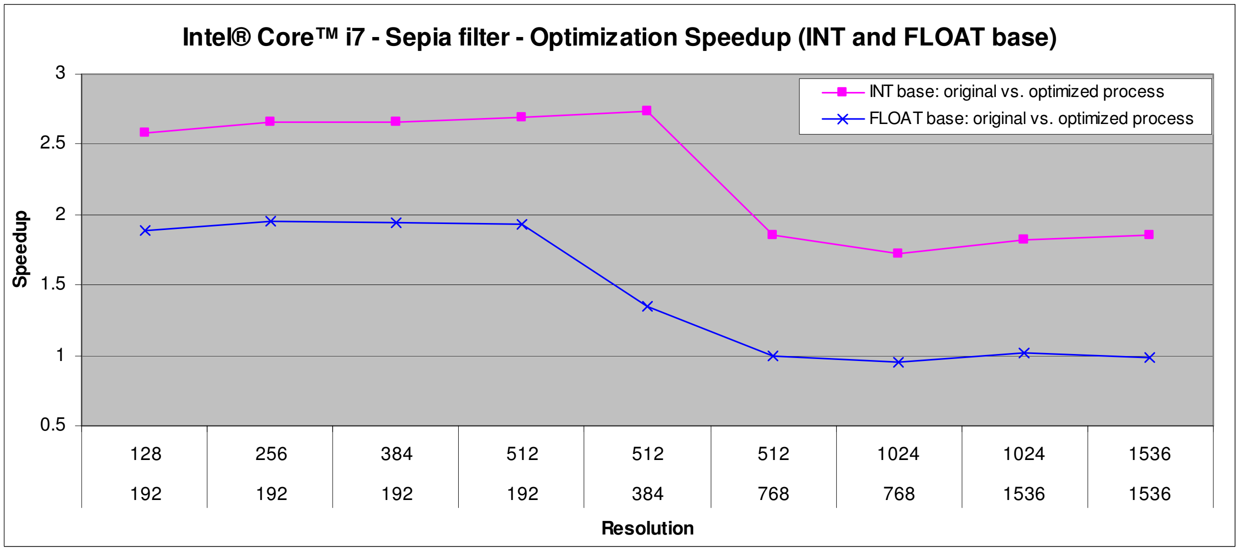


Figure 2: Sepia filter performance comparison. Intel® Core™ i7 processor

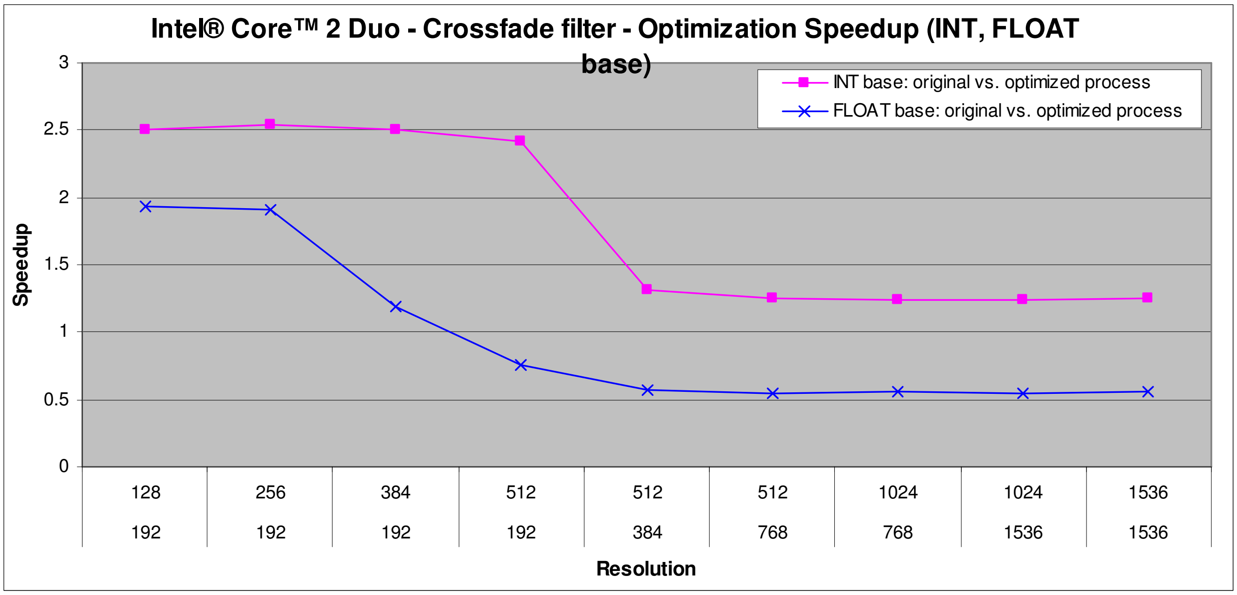


Figure 3: Crossfade filter performance comparison. Intel® Core™ Duo processor

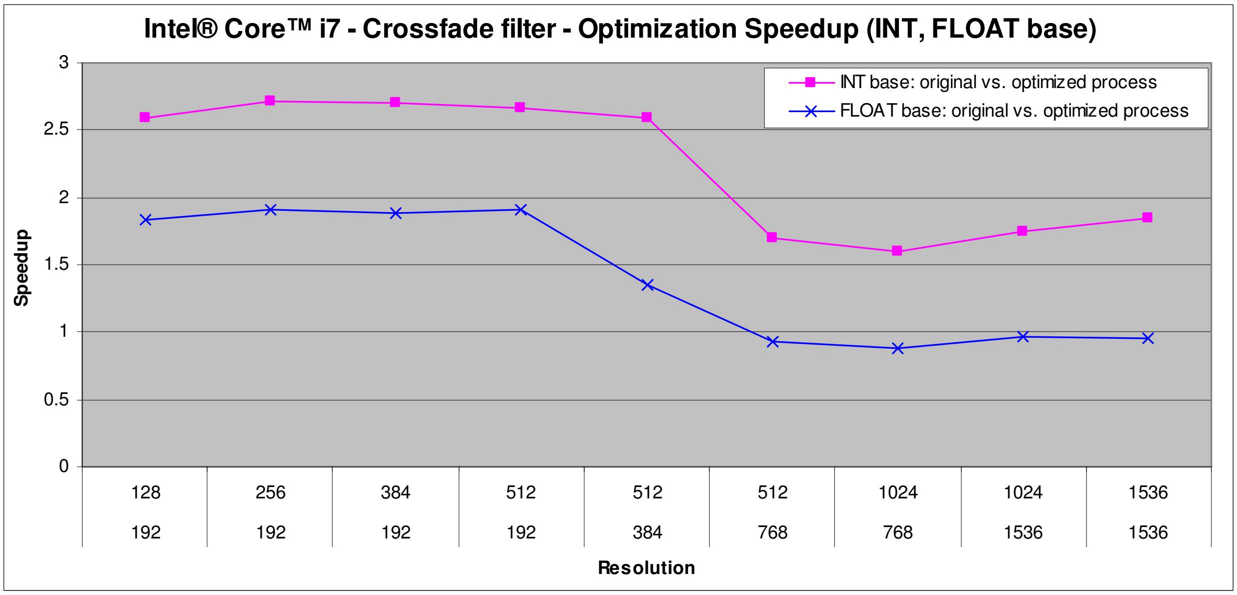


Figure 4: Crossfade filter performance comparison. Intel® Core™ i7 processor

Intel® AVX optimization results

The following graph shows the estimated Intel® AVX performance by using Intel® AVX simulator. For instance, the Crossfade filter using Intel® AVX provides at best a 34% speedup over the Intel® SSE implementation. Note that the simulated Intel® AVX performance only takes into account the speedup from using the new instruction set, performance benefits for platform architectural improvements are not included.

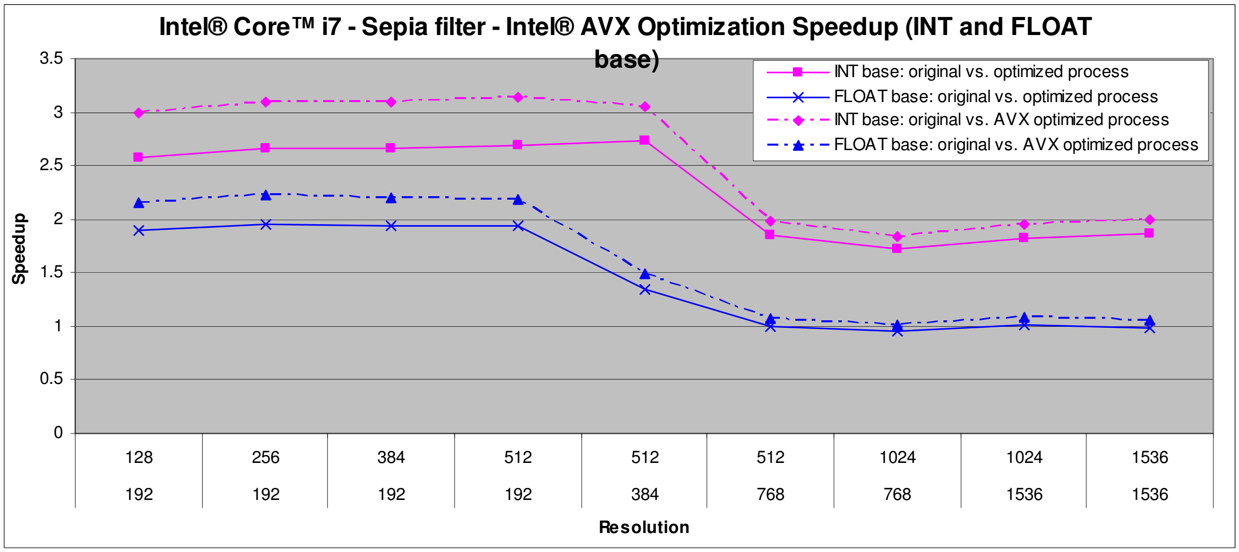


Figure 5: Sepia filter Intel® AVX performance estimate

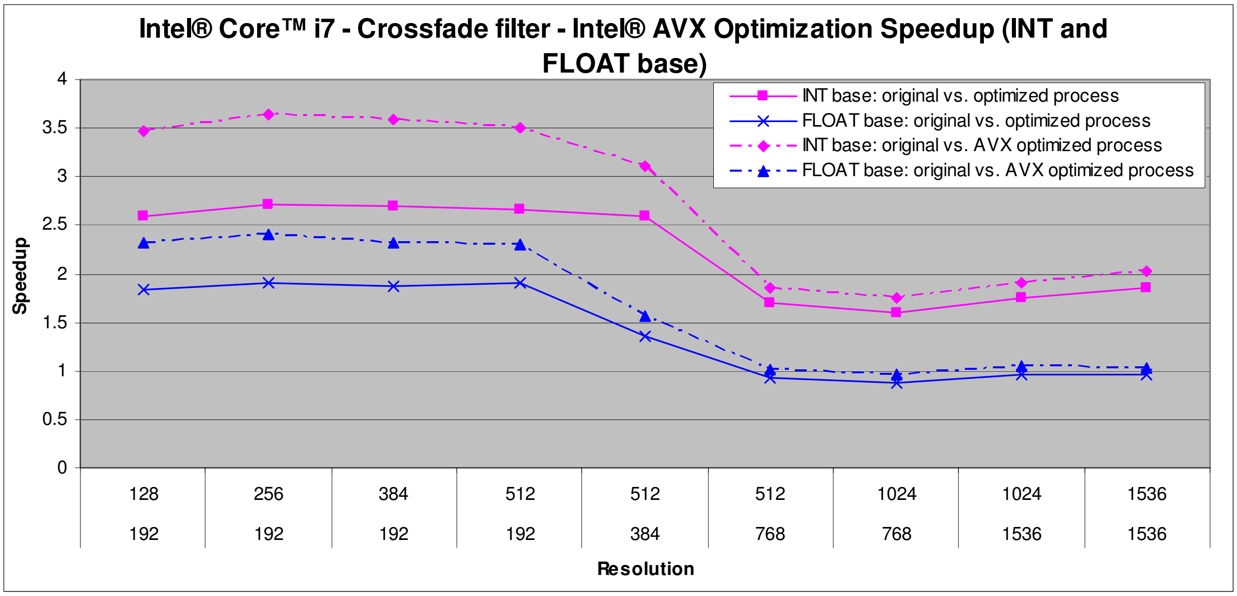


Figure 6: Crossfade filter Intel® AVX performance estimate

An alternate way of exploring at the filter processing efficiency is to compare the number of processor clock cycles used per pixel processed for the un-optimized process versus the Intel® SSE and Intel® AVX optimized process.

From the below graph it it clear that using the proposed techniques the efficiency is improved significantly. For instance, the Intel® AVX optimized Sepia filter (integer base) processing uses 72% less clock cycles to process the same amount of pixels compared to baseline.

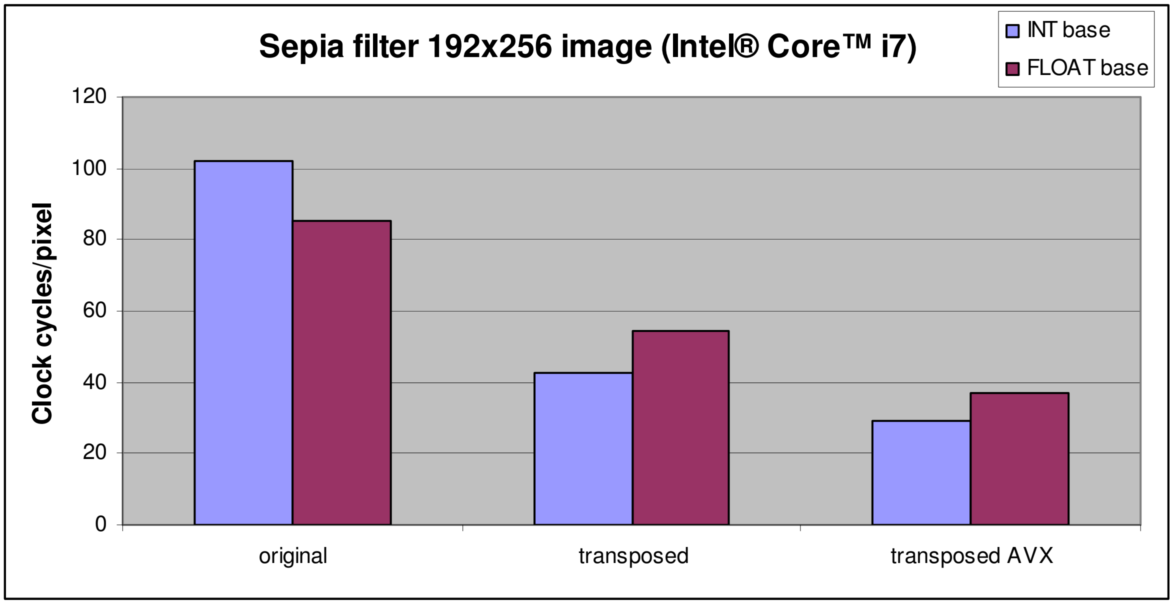


Figure 7: Clock cycle/pixel comparison. Intel® Core™ i7 processor

Conclusion

From the measured performance it is clear that utilizing the proposed techniques we can achieve good performance speedup. Below table summarizes observed (and Intel® AVX simulated) performance speedup for on the Intel® Core™ i7 processor platform. Recommended chunk size is ~50000 pixels (or 256x192), but as can be observed from the graphs, comparable speedup, for the analyzed filters, can be seen in the range from 128x192 up to 512x192 for both integer and float bases.

Filter	**Intel® SSE Speedup**	**Intel® AVX Speedup**
Sepia (int base)	2.6x	3.1x
Sepia (float base)	1.9x	2.2x
Crossfade (int base)	2.7x	3.6x
Crossfade (float base)	1.9x	2.4x

Table 4: Observed and estimated Intel® AVX speedup using 256x192 chunk size

For more complex algorithms, even greater speedup might be attainable due to less relative overhead of the transformation functions as well as potentially lower impact of memory access bottlenecks.

Other algorithms using this technique should determine the optimal chunk size for the targeted platform to achieve optimal performance.

Note that significant additional performance speedup might be obtained by applying multi threading techniques using data decomposition. Also note that the performance of the filters, now developed to take advantage of autovectorization, could most likely be improved by hand coding using intrinsics or assembler.

Future Intel® architectures such as Intel® discrete graphics will provide additional performance speedup opportunities, through extended SIMD support. This topic and other image processing algorithms such as HDR may be addressed in future papers.

References

- Intel® AVX - Home Page: http://software.intel.com/en-us/avx/
- Intel® AVX Programming Reference: http://software.intel.com/en-us/avx/ → Intel® Advanced Vector Extensions Programming Reference
- Intel® Architecture Code Analyzer: http://software.intel.com/en-us/articles/intel-architecture-code-analyzer/
- Intel® Software Development Emulator: http://software.intel.com/en-us/articles/intel-software-development-emulator/
- Intel® Compilers: http://software.intel.com/en-us/intel-compilers/

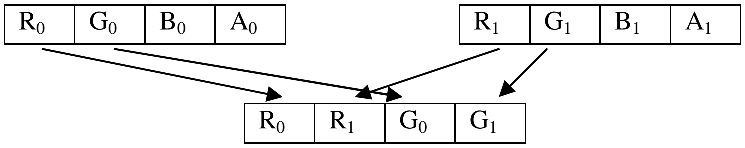
Appendix A

Transpose float image data from AoS <-> SoA

The following function uses Intel® SSE intrinsics for efficient transposing of image data, processing 4 single precision float RGBA pixels (16 floats, 64 bytes) at a time. The transposition is achieved in two steps (aside from loads and stores).

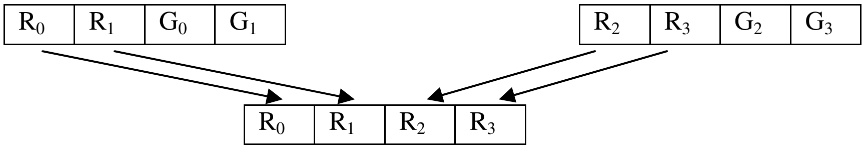
(1) Unpack high and low float values from source data using UNPCKLPS and UNPCKHPS. For instance the first UNPCKLPS operates on the first two pixels in the data structure

```
mm_unpacklo_ps(_mm_load_ps(pSrc+i), _mm_load_ps(pSrc+i+4)
```



(2) Move high and low values of same color channel into destination using MOVLHPS and MOVHLPS. For instance the first MOVLHPS operates on two of the unpacked registers to extract the red color plane values into one register

```
_mm_movelh_ps(vfT0, vfT2)
```



```
void transposeFloat(float *pSrc, float *pDst, unsigned int imageSize)
{
        __m128 vfT0, vfP0, vfT2, vfP2;

        for(unsigned int i=0;i<imageSize*4;i+=16)
        {
                vfT0 = _mm_unpacklo_ps(_mm_load_ps(pSrc+i),   _mm_load_ps(pSrc+i+4));  // R0R1G0G1
                vfP0 = _mm_unpackhi_ps(_mm_load_ps(pSrc+i),   _mm_load_ps(pSrc+i+4));  // B0B1A0A1
                vfT2 = _mm_unpacklo_ps(_mm_load_ps(pSrc+i+8), _mm_load_ps(pSrc+i+12)); // R2R3G2G3
                vfP2 = _mm_unpackhi_ps(_mm_load_ps(pSrc+i+8), _mm_load_ps(pSrc+i+12)); // B2B3A2A3

                _mm_store_ps(pDst+i,    _mm_movelh_ps(vfT0, vfT2)); // R0R1R2R3
                _mm_store_ps(pDst+i+4,  _mm_movehl_ps(vfT2, vfT0)); // G0G1G2G3
                _mm_store_ps(pDst+i+8,  _mm_movelh_ps(vfP0, vfP2)); // B0B1B2B3
                _mm_store_ps(pDst+i+12, _mm_movehl_ps(vfP2, vfP0)); // A0A1A2A3
        }
}
```

Note that the function loop core compiles into four implicit loads to intermediate registers.

The same function can be used to invert the transpose.

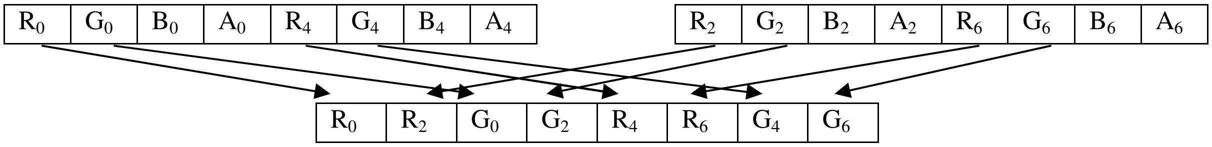
For processors supporting Intel® AVX we can efficiently operate on 8 single precision float RGBA pixels (32 floats, 128 bytes) at a time. The following function implements efficient transposition using the new Intel® AVX instructions. Transposition is achieved in three steps (aside from loads and stores).

(1) Using VPERM2F128, rearrange 8 pixels loaded from the AoS data structure into order suitable for final ordered SoA results after unpack. For instance, the first VPERM2F128 operates on pixel 0,1,4,5 in the data structure rearranging (using mask=00100000) the order to register containing pixel 0,4.

```
_mm256_permute2f128_ps(ld0, ld2, 0x20)
```

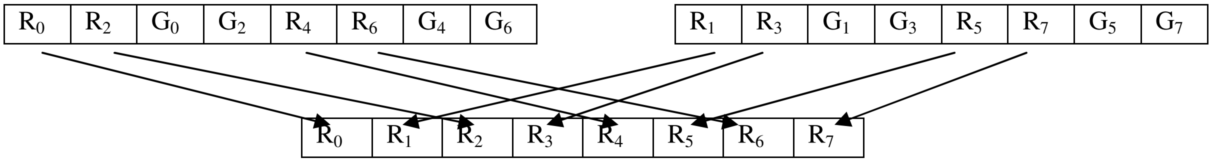
(2) Unpack hi and low float values using VUNPCKLPS and VUNPCKHPS. For instance VUNPCKLPS operates on the 4 pixels in the rearranged registers.

```
_mm256_unpacklo_ps(pm0, pm1)
```


R_0 G_0 B_0 A_0 R_4 G_4 B_4 A_4
R_2 G_2 B_2 A_2 R_6 G_6 B_6 A_6
R_0 R_2 G_0 G_2 R_4 R_6 G_4 G_6

(3) Again unpack hi and low float values using VUNPCKLPS and VUNPCKHPS. For instance VUNPCKLPS operates on the 4 pixels in the previously unpacked registers to produce the red color channel SoA register

```
_mm256_unpacklo_ps(up0, up2)
```


R_0 R_2 G_0 G_2 R_4 R_6 G_4 G_6
R_1 R_3 G_1 G_3 R_5 R_7 G_5 G_7
R_0 R_1 R_2 R_3 R_4 R_5 R_6 R_7

```
void transposeFloatAVX(float *pSrc, float *pDst, unsigned int imageSize)
{
        __m256 ld0, ld1, ld2, ld3;
        __m256 pm0, pm1, pm2, pm3;
        __m256 up0, up1, up2, up3;

        for(unsigned int i=0;i<imageSize*4;i+=32)
        {
                ld0 = _mm256_load_ps(pSrc+i);
                ld1 = _mm256_load_ps(pSrc+i+8);
                ld2 = _mm256_load_ps(pSrc+i+16);
                ld3 = _mm256_load_ps(pSrc+i+24);

                pm0 = _mm256_permute2f128_ps(ld0, ld2, 0x20); // R0G0B0A0 R4G4B4A4
                pm1 = _mm256_permute2f128_ps(ld1, ld3, 0x20); // R2G2B2A2 R6G6B6A6
                pm2 = _mm256_permute2f128_ps(ld0, ld2, 0x31); // R1G1B1A1 R5G5B5A5
                pm3 = _mm256_permute2f128_ps(ld1, ld3, 0x31); // R3G3B3A3 R7G7B7A7

                up0 = _mm256_unpacklo_ps(pm0, pm1);  // R0R2G0G2 R4R6G4G6
                up1 = _mm256_unpackhi_ps(pm0, pm1);  // B0B2A0A2 B4B6A4A6
                up2 = _mm256_unpacklo_ps(pm2, pm3);  // R1R3G1G3 R5R7G5G7
                up3 = _mm256_unpackhi_ps(pm2, pm3);  // B1B3A1A3 B5B7A5A7

                _mm256_store_ps(pDst+i,    _mm256_unpacklo_ps(up0, up2)); // R0R1R2R3 R4R5R6R7
                _mm256_store_ps(pDst+i+8,  _mm256_unpackhi_ps(up0, up2)); // G0G1G2G3 G4G5G6G7
                _mm256_store_ps(pDst+i+16, _mm256_unpacklo_ps(up1, up3)); // B0B1B2B3 B4B5B6B7
                _mm256_store_ps(pDst+i+24, _mm256_unpackhi_ps(up1, up3)); // A0A1A2A3 A4A5A6A7
        }
}
```

Consider that the amount of data processed is 2x compared to the Intel® SSE transpose implementation.

Since the Intel® AVX version of the transposition function is not self inverse it can not be used to perform inverse transposition. Instead the following equivalent function was created.

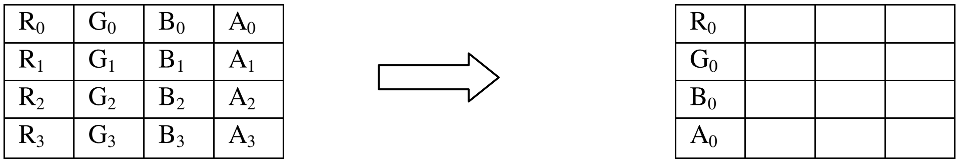
```
void transposeFloatAVX_reverse(float *pSrc, float *pDst, unsigned int imageSize)
{
        __m256 ld0, ld1, ld2, ld3;
        __m256 up0, up1, up2, up3;
        __m256 upb0, upb1, upb2, upb3;

        for(unsigned int i=0;i<imageSize*4;i+=32)
        {
                ld0 = _mm256_load_ps(pSrc+i);
                ld1 = _mm256_load_ps(pSrc+i+8);
                ld2 = _mm256_load_ps(pSrc+i+16);
                ld3 = _mm256_load_ps(pSrc+i+24);

                up0 = _mm256_unpacklo_ps(ld0, ld2);  // R0B0R1B1 R4B4R5B5
                up1 = _mm256_unpackhi_ps(ld0, ld2);  // R2B2R3B3 R6B6R7B7
                up2 = _mm256_unpacklo_ps(ld1, ld3);  // G0A0G1A1 G4A4G5A5
                up3 = _mm256_unpackhi_ps(ld1, ld3);  // G2A2G3A3 G6A6G7A7

                upb0 = _mm256_unpacklo_ps(up0, up2); // R0G0B0A0 R4G4B4A4
                upb1 = _mm256_unpackhi_ps(up0, up2); // R1G1B1A1 R5G5B5A5
                upb2 = _mm256_unpacklo_ps(up1, up3); // R2G2B2A2 R6G6B6A6
                upb3 = _mm256_unpackhi_ps(up1, up3); // R3G3B3A3 R7G7B7A7

                // R0G0B0A0 R1G1B1A1
                _mm256_store_ps(pDst+i,    _mm256_permute2f128_ps(upb0, upb1, 0x20));
                // R2G2B2A2 R3G3B3A3
                _mm256_store_ps(pDst+i+8,  _mm256_permute2f128_ps(upb2, upb3, 0x20));
                // R4G4B4A4 R5G5B5A5
                _mm256_store_ps(pDst+i+16, _mm256_permute2f128_ps(upb0, upb1, 0x31));
                // R6G6B6A6 R7G7B7A7
                _mm256_store_ps(pDst+i+24, _mm256_permute2f128_ps(upb2, upb3, 0x31));
        }
}
```

Expand image data from integer to float

The following function efficiently expands 8 bit integer image data into 32 bit float image data, using Intel® SSE intrinsics. The expansion is achieved in two steps (aside from loads and stores).

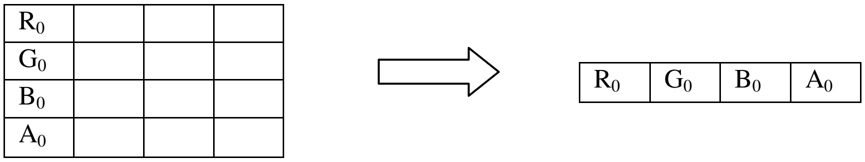
(1) Using the PSHUFB instruction, shuffle 8 bit integer values according to the four masks. This effectively performs expansion into 32 bit integer values. For instance the first PSHUFB with mask "`SHUFFLEMASKY1`" operates on the 16 integer values in the following way

`_mm_shuffle_epi8(ipm, sm1)`



(2) Convert 32 bit integer values to 32 bit float values, using CVTDQ2PS, as in the first convert illustrated below

`_mm_cvtepi32_ps(_mm_shuffle_epi8(ipm, sm1))`



```
__m128i SHUFFLEMASKY1 = {     0x00, 0x80, 0x80, 0x80,
                              0x01, 0x80, 0x80, 0x80,
                              0x02, 0x80, 0x80, 0x80,
                              0x03, 0x80, 0x80, 0x80};
__m128i SHUFFLEMASKY2 = {     0x04, 0x80, 0x80, 0x80,
                              0x05, 0x80, 0x80, 0x80,
                              0x06, 0x80, 0x80, 0x80,
                              0x07, 0x80, 0x80, 0x80};
__m128i SHUFFLEMASKY3 = {     0x08, 0x80, 0x80, 0x80,
                              0x09, 0x80, 0x80, 0x80,
                              0x0A, 0x80, 0x80, 0x80,
                              0x0B, 0x80, 0x80, 0x80};
__m128i SHUFFLEMASKY4 = {     0x0C, 0x80, 0x80, 0x80,
                              0x0D, 0x80, 0x80, 0x80,
                              0x0E, 0x80, 0x80, 0x80,
                              0x0F, 0x80, 0x80, 0x80};
```

```
void expandIntToFloat(unsigned char *pSrc, float *pDst, unsigned int imageSize)
{
        __m128i ipm;
        __m128i sm1, sm2, sm3, sm4;
        sm1 = _mm_load_si128(&SHUFFLEMASKY1);
        sm2 = _mm_load_si128(&SHUFFLEMASKY2);
        sm3 = _mm_load_si128(&SHUFFLEMASKY3);
        sm4 = _mm_load_si128(&SHUFFLEMASKY4);
        for(unsigned int i=0;i<imageSize*4;i+=16)
        {
                ipm = _mm_load_si128((__m128i *)(pSrc+i));
                _mm_store_ps(pDst+i,    _mm_cvtepi32_ps(_mm_shuffle_epi8(ipm, sm1)));
                _mm_store_ps(pDst+i+4,  _mm_cvtepi32_ps(_mm_shuffle_epi8(ipm, sm2)));
                _mm_store_ps(pDst+i+8,  _mm_cvtepi32_ps(_mm_shuffle_epi8(ipm, sm3)));
                _mm_store_ps(pDst+i+12, _mm_cvtepi32_ps(_mm_shuffle_epi8(ipm, sm4)));
        }
}
```

Note that the function loop core compiles info three implicit loads for intermediate registers.

For processors supporting Intel® AVX we can efficiently operate on 8 single precision float RGBA pixels (32 floats, 128 bytes) at a time. The expansion is achieved in two steps as in the above Intel® SSE implementation.

```
void expandIntToFloatAVX(unsigned char *pSrc, float *pDst, unsigned int imageSize)
{
        __m128i ld0, ld1;
        __m128i sm1, sm2, sm3, sm4;
        sm1 = _mm_load_si128(&SHUFFLEMASKY1);
        sm2 = _mm_load_si128(&SHUFFLEMASKY2);
        sm3 = _mm_load_si128(&SHUFFLEMASKY3);
        sm4 = _mm_load_si128(&SHUFFLEMASKY4);
        for(unsigned int i=0;i<imageSize*4;i+=32)
        {
                ld0 = _mm_load_si128((__m128i *)(pSrc+i));
                ld1 = _mm_load_si128((__m128i *)(pSrc+i+16));
                _mm_store_ps(pDst+i,    _mm_cvtepi32_ps(_mm_shuffle_epi8(ld0, sm1)));
                _mm_store_ps(pDst+i+4,  _mm_cvtepi32_ps(_mm_shuffle_epi8(ld0, sm2)));
                _mm_store_ps(pDst+i+8,  _mm_cvtepi32_ps(_mm_shuffle_epi8(ld0, sm3)));
                _mm_store_ps(pDst+i+12, _mm_cvtepi32_ps(_mm_shuffle_epi8(ld0, sm4)));
                _mm_store_ps(pDst+i+16, _mm_cvtepi32_ps(_mm_shuffle_epi8(ld1, sm1)));
                _mm_store_ps(pDst+i+20, _mm_cvtepi32_ps(_mm_shuffle_epi8(ld1, sm2)));
                _mm_store_ps(pDst+i+24, _mm_cvtepi32_ps(_mm_shuffle_epi8(ld1, sm3)));
                _mm_store_ps(pDst+i+28, _mm_cvtepi32_ps(_mm_shuffle_epi8(ld1, sm4)));
        }
}
```

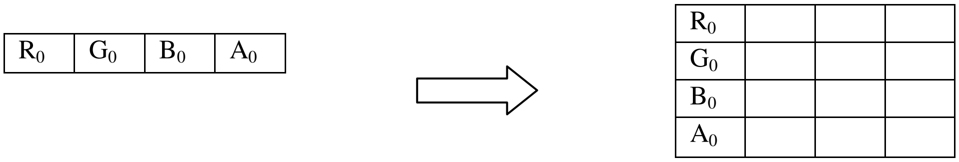
As can be seen the AVX implementation of the expansion routine does not use any Intel® AVX instructions. The reason for this is due to the fact that there are no 256-bit equivalent instructions to 8 bit integer PSHUFB or POR. Some Intel® AVX instructions could be used but they would have to mixed with Intel® SSE with additional extract operations. We found that implementation using Intel® AVX mixed with Intel® SSE did not render improved performance.

Compress image data from float to integer

The following function compresses 32 bit float image data into 8 bit integer image data, using Intel® SSE intrinsics. The compression is achieved in three steps (aside from loads and stores).

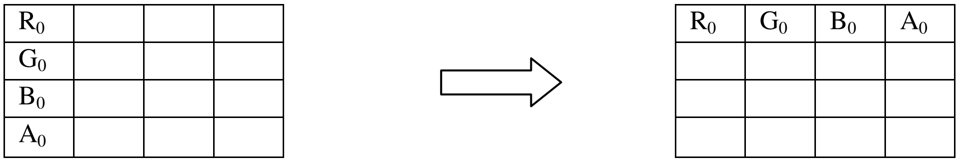
(1) Convert 32 bit float values to 32 bit integer values, using CVTPS2DQ, as in the first convert illustrated below

```
_mm_cvtps_epi32(_mm_load_ps(pSrc+i))
```



(2) Shuffle integer, using PSHUFB, values to effectively perform compression into 8 bit integer values as in the first PSHUFB operation using the “SHUFFLEMASKYC1” mask

```
_mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i)),    sm1)
```



(3) Perform bitwise “or” (using the POR instruction) to merge the 4 registers (pixel rows) into register containing all 16 integers.

```
_mm_or_si128(res1, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+4)),  sm2))
```

```
__m128i SHUFFLEMASKYC1 = {    0x00, 0x04, 0x08, 0x0C,
                              0x80, 0x80, 0x80, 0x80,
                              0x80, 0x80, 0x80, 0x80,
                              0x80, 0x80, 0x80, 0x80};
__m128i SHUFFLEMASKYC2 = {    0x80, 0x80, 0x80, 0x80,
                              0x00, 0x04, 0x08, 0x0C,
                              0x80, 0x80, 0x80, 0x80,
                              0x80, 0x80, 0x80, 0x80};
__m128i SHUFFLEMASKYC3 = {    0x80, 0x80, 0x80, 0x80,
                              0x80, 0x80, 0x80, 0x80,
                              0x00, 0x04, 0x08, 0x0C,
                              0x80, 0x80, 0x80, 0x80};
__m128i SHUFFLEMASKYC4 = {    0x80, 0x80, 0x80, 0x80,
                              0x80, 0x80, 0x80, 0x80,
                              0x80, 0x80, 0x80, 0x80,
                              0x00, 0x04, 0x08, 0x0C};
```

```
void compressFloatToInt(float *pSrc, unsigned char *pDst, unsigned int imageSize)
{
   __m128i res1, res2;
   __m128i sm1, sm2, sm3, sm4;
   sm1 = _mm_load_si128(&SHUFFLEMASKYC1);
   sm2 = _mm_load_si128(&SHUFFLEMASKYC2);
   sm3 = _mm_load_si128(&SHUFFLEMASKYC3);
   sm4 = _mm_load_si128(&SHUFFLEMASKYC4);
   for(unsigned int i=0;i<imageSize*4;i+=16)
   {
      res1 =                    _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i)),    sm1);
      res1 = _mm_or_si128(res1, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+4)),  sm2));
      res2 =                    _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+8)),  sm3);
      res2 = _mm_or_si128(res2, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+12)), sm4));
      _mm_store_si128((__m128i *)(pDst+i), _mm_or_si128(res1, res2));
   }
}
```

For processors supporting Intel® AVX we can efficiently operate on 8 single precision float RGBA pixels (32 floats, 128 bytes) at a time. The compression is achieved in three steps as in the Intel® SSE implementation.

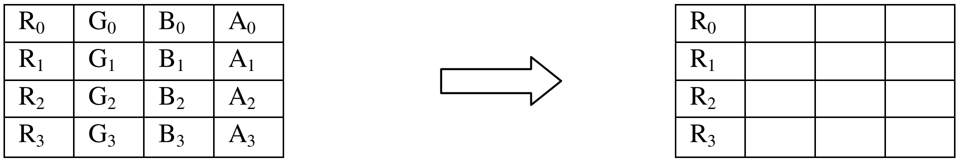
```
void compressFloatToIntAVX(float *pSrc, unsigned char *pDst, unsigned int imageSize)
{
   __m128i res0, res1, res2, res3;
   __m128i sm1, sm2, sm3, sm4;
   sm1 = _mm_load_si128(&SHUFFLEMASKYC1);
   sm2 = _mm_load_si128(&SHUFFLEMASKYC2);
   sm3 = _mm_load_si128(&SHUFFLEMASKYC3);
   sm4 = _mm_load_si128(&SHUFFLEMASKYC4);
   for(unsigned int i=0;i<imageSize*4;i+=32)
   {
      res0 =                    _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i)),    sm1);
      res0 = _mm_or_si128(res0, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+4)),  sm2));
      res1 =                    _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+8)),  sm3);
      res1 = _mm_or_si128(res1, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+12)), sm4));
      res0 = _mm_or_si128(res0, res1);

      res2 =                    _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+16)), sm1);
      res2 = _mm_or_si128(res2, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+20)), sm2));
      res3 =                    _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+24)), sm3);
      res3 = _mm_or_si128(res3, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+28)), sm4));
      res2 = _mm_or_si128(res2, res3);

      _mm_store_si128((__m128i *)(pDst+i),    res0);
      _mm_store_si128((__m128i *)(pDst+i+16), res2);
   }
}
```

Transpose integer image data with expansion to float

The following function performs a combined expansion and transpose of an AoS 8 bit integer image data into a SoA 32 bit float image data, using Intel® SSE intrinsics. The transpose/expansion is achieved in two steps (aside from loads and stores).

(1) Using PSHUFB instruction, shuffle 8 bit integer values according to the four masks. This effectively performs the expansion with reordering into 32 bit integer values. For instance the first PSHUFB using mask "`SHUFFLEMASKX1`" operates on 16 integer values in the following way

`_mm_shuffle_epi8(ipm, sm1)`

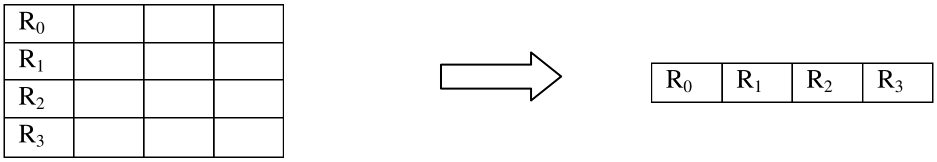


R_0	G_0	B_0	A_0
R_1	G_1	B_1	A_1
R_2	G_2	B_2	A_2
R_3	G_3	B_3	A_3

R_0			
R_1			
R_2			
R_3			

(2) Convert 32 bit integer values to 32 bit float values, using CVTDQ2PS, as in the first convert operation illustrated

`_mm_cvtepi32_ps(_mm_shuffle_epi8(ipm, sm1))`



R_0			
R_1			
R_2			
R_3			

R_0	R_1	R_2	R_3

```
__m128i SHUFFLEMASKX1 = {      0x00, 0x80, 0x80, 0x80,
                               0x04, 0x80, 0x80, 0x80,
                               0x08, 0x80, 0x80, 0x80,
                               0x0C, 0x80, 0x80, 0x80};
__m128i SHUFFLEMASKX2 = {      0x01, 0x80, 0x80, 0x80,
                               0x05, 0x80, 0x80, 0x80,
                               0x09, 0x80, 0x80, 0x80,
                               0x0D, 0x80, 0x80, 0x80};
__m128i SHUFFLEMASKX3 = {      0x02, 0x80, 0x80, 0x80,
                               0x06, 0x80, 0x80, 0x80,
                               0x0A, 0x80, 0x80, 0x80,
                               0x0E, 0x80, 0x80, 0x80};
__m128i SHUFFLEMASKX4 = {      0x03, 0x80, 0x80, 0x80,
                               0x07, 0x80, 0x80, 0x80,
                               0x0B, 0x80, 0x80, 0x80,
                               0x0F, 0x80, 0x80, 0x80};
```

```
void transposeAndExpandIntToFloat(unsigned char *pSrc, float *pDst, unsigned int imageSize)
{
        __m128i ipm;
        __m128i sm1, sm2, sm3, sm4;
        sm1 = _mm_load_si128(&SHUFFLEMASKX1);
        sm2 = _mm_load_si128(&SHUFFLEMASKX2);
        sm3 = _mm_load_si128(&SHUFFLEMASKX3);
        sm4 = _mm_load_si128(&SHUFFLEMASKX4);
        for(unsigned int i=0;i<imageSize*4;i+=16)
        {
                ipm = _mm_load_si128((__m128i *)(pSrc+i));
                _mm_store_ps(pDst+i,    _mm_cvtepi32_ps(_mm_shuffle_epi8(ipm, sm1)));
                _mm_store_ps(pDst+i+4,  _mm_cvtepi32_ps(_mm_shuffle_epi8(ipm, sm2)));
                _mm_store_ps(pDst+i+8,  _mm_cvtepi32_ps(_mm_shuffle_epi8(ipm, sm3)));
                _mm_store_ps(pDst+i+12, _mm_cvtepi32_ps(_mm_shuffle_epi8(ipm, sm4)));
        }
}
```

Note that the function loop core compiles info three implicit loads for intermediate registers.

For processors supporting Intel® AVX we can efficiently operate on 8 single precision float RGBA pixels (32 floats, 128 bytes) at a time. The transpose/expansion is achieved in two steps as in the Intel® SSE implementation.

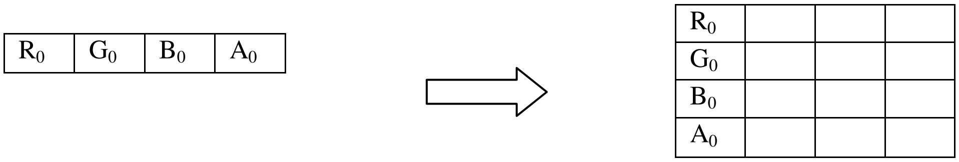
```
void transposeAndExpandIntToFloatAVX(unsigned char *pSrc, float *pDst, unsigned int imageSize)
{
        __m128i ld0, ld1;
        __m128i sm1, sm2, sm3, sm4;
        sm1 = _mm_load_si128(&SHUFFLEMASKX1);
        sm2 = _mm_load_si128(&SHUFFLEMASKX2);
        sm3 = _mm_load_si128(&SHUFFLEMASKX3);
        sm4 = _mm_load_si128(&SHUFFLEMASKX4);
        for(unsigned int i=0;i<imageSize*4;i+=32)
        {
                ld0 = _mm_load_si128((__m128i *)(pSrc+i));
                ld1 = _mm_load_si128((__m128i *)(pSrc+i+16));
                _mm_store_ps(pDst+i,    _mm_cvtepi32_ps(_mm_shuffle_epi8(ld0, sm1)));
                _mm_store_ps(pDst+i+4,  _mm_cvtepi32_ps(_mm_shuffle_epi8(ld1, sm1)));
                _mm_store_ps(pDst+i+8,  _mm_cvtepi32_ps(_mm_shuffle_epi8(ld0, sm2)));
                _mm_store_ps(pDst+i+12, _mm_cvtepi32_ps(_mm_shuffle_epi8(ld1, sm2)));
                _mm_store_ps(pDst+i+16, _mm_cvtepi32_ps(_mm_shuffle_epi8(ld0, sm3)));
                _mm_store_ps(pDst+i+20, _mm_cvtepi32_ps(_mm_shuffle_epi8(ld1, sm3)));
                _mm_store_ps(pDst+i+24, _mm_cvtepi32_ps(_mm_shuffle_epi8(ld0, sm4)));
                _mm_store_ps(pDst+i+28, _mm_cvtepi32_ps(_mm_shuffle_epi8(ld1, sm4)));
        }
}
```

Transpose float image data with compression to int

The following function performs a combined compression and transpose of an AoS 32 bit float image data into a SoA 8 bit integer image data, using Intel® SSE intrinsics. The compression/transpose is achieved in three steps (aside from loads and stores).

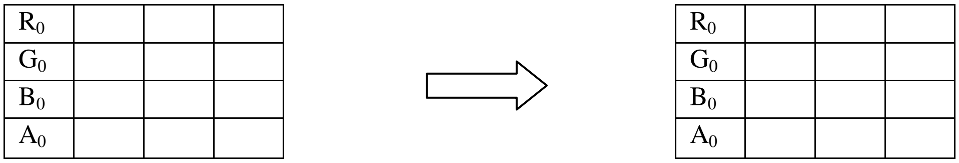
(1) Convert 32 bit float values to 32 bit integer values, using CVTPS2DQ, as in the first convert illustrated below

`_mm_cvtps_epi32(_mm_load_ps(pSrc+i))`



(2) Using PSHUFB instruction, shuffle integer values to effectively perform compression into 8 bit integer values as in the first PSHUFB operation using the "`SHUFFLEMASKXC1`" mask

`_mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i)),    sm1)`



(3) Perform bitwise "or" (using the POR instruction) to merge the 4 registers (pixel columns) into register containing all 16 integers

`_mm_or_si128(res0, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+4)),  sm2))`

```
__m128i SHUFFLEMASKXC1 = {    0x00, 0x80, 0x80, 0x80,
                              0x04, 0x80, 0x80, 0x80,
                              0x08, 0x80, 0x80, 0x80,
                              0x0C, 0x80, 0x80, 0x80};
__m128i SHUFFLEMASKXC2 = {    0x80, 0x00, 0x80, 0x80,
                              0x80, 0x04, 0x80, 0x80,
                              0x80, 0x08, 0x80, 0x80,
                              0x80, 0x0C, 0x80, 0x80};
__m128i SHUFFLEMASKXC3 = {    0x80, 0x80, 0x00, 0x80,
                              0x80, 0x80, 0x04, 0x80,
                              0x80, 0x80, 0x08, 0x80,
                              0x80, 0x80, 0x0C, 0x80};
__m128i SHUFFLEMASKXC4 = {    0x80, 0x80, 0x80, 0x00,
                              0x80, 0x80, 0x80, 0x04,
                              0x80, 0x80, 0x80, 0x08,
                              0x80, 0x80, 0x80, 0x0C};
```

```
void transposeAndCompressToInt(float *pSrc, unsigned char *pDst, unsigned int imageSize)
{
   __m128i res0, res1;
   __m128i sm1, sm2, sm3, sm4;
   sm1 = _mm_load_si128(&SHUFFLEMASKXC1);
   sm2 = _mm_load_si128(&SHUFFLEMASKXC2);
   sm3 = _mm_load_si128(&SHUFFLEMASKXC3);
   sm4 = _mm_load_si128(&SHUFFLEMASKXC4);
   for(unsigned int i=0;i<imageSize*4;i+=16)
   {
      res0 =                    _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i)),    sm1);
      res0 = _mm_or_si128(res0, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+4)),  sm2));
      res1 =                    _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+8)),  sm3);
      res1 = _mm_or_si128(res1, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+12)), sm4));
      _mm_store_si128((__m128i *)(pDst+i), _mm_or_si128(res0, res1));
   }
}
```

For processors supporting Intel® AVX we can efficiently operate on 8 single precision float RGBA pixels (32 floats, 128 bytes) at a time. The compression/transpose is achieved in three steps as in the Intel® SSE implementation.

```
void transposeAndCompressToIntAVX(float *pSrc, unsigned char *pDst, unsigned int imageSize)
{
   __m128i res0, res1, res2, res3;
   __m128i sm1, sm2, sm3, sm4;
   sm1 = _mm_load_si128(&SHUFFLEMASKXC1);
   sm2 = _mm_load_si128(&SHUFFLEMASKXC2);
   sm3 = _mm_load_si128(&SHUFFLEMASKXC3);
   sm4 = _mm_load_si128(&SHUFFLEMASKXC4);
   for(unsigned int i=0;i<imageSize*4;i+=32)
   {
      res0 =                    _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i)),    sm1);
      res0 = _mm_or_si128(res0, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+8)),  sm2));
      res1 =                    _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+16)), sm3);
      res1 = _mm_or_si128(res1, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+24)), sm4));
      res0 = _mm_or_si128(res0, res1);

      res2 =                    _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+4)),  sm1);
      res2 = _mm_or_si128(res2, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+12)), sm2));
      res3 =                    _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+20)), sm3);
      res3 = _mm_or_si128(res3, _mm_shuffle_epi8(_mm_cvtps_epi32(_mm_load_ps(pSrc+i+28)), sm4));
      res2 = _mm_or_si128(res2, res3);

      _mm_store_si128((__m128i *)(pDst+i),    res0);
      _mm_store_si128((__m128i *)(pDst+i+16), res2);
   }
}
```